CSC263 Assignment #3

You are allowed to work in a group of at most 3 members, and make one submission as a group. You should clearly write names of all group members in the first page of your submission. Everyone in the same group will get the same marks.
Please read and understand the policy on Academic Honesty on the course website. Then, to protect yourself, list on the front of your submission every source of information you used to complete this homework (other than your own lecture and tutorial notes). For example, indicate clearly the name of every student with whom you had discussions, the title and sections of every textbook you consulted (including the course textbook), the source of every web document you used (including documents from the course webpage), etc.
For each question, please write up detailed answers carefully. Make sure that you use notation and terminology correctly, and that you explain and justify what you are doing. Marks will be deducted for incorrect or ambiguous use of notation and terminology, and for making incorrect, unjustified, ambiguous, or vague claims in your solutions.

From this assignment on, most questions on writing algorithms will not explicitly require pseudocode. You may either describe in english, or write pseudocode, or give both (for example, write pseudocode for sub-procedures). When you describe your algorithm, write concisely and precisely in a few lines; you may also itemize the algorithm in several steps.
We will also give some simple exercise questions, which are similar to the ones in the midterm.

1. (*5 points*) Linear probing in open-addressing hash table.
Consider an open-addressing hash table that uses the hash function

$h(k, i) = (h'(k) + i) \mod 13$,
$h'(k) = k \mod 13$.

Fill in the slots below by inserting the following sequence of keys to the hash table defined above:

- Already inserted: 10, 8, 130, 13
- To insert: 27, 9, 14, 23, 21

slot	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
key	130	13							8		10		

2. (*5 points*) AVL tree insertion and deletion.
(a) Draw the AVL tree after inserting the following sequence of keys into an empty tree:
5, 8, 7, 1, 2
(b) Draw the AVL tree after deleting 8 from the AVL tree obtained in (a).

3. (*10 points*) Amortized analysis of deletion in dynamic arrays using the potential method.
Suppose that instead of shrinking a table by halving its size when its load factor drops below 1/4, we shrink it by multiplying its size by 2/3 when its load factor drops below 1/3. Using the potential function

$\Phi(T) = |\, 2\, T.num - T.size \,|$,

show that the amortized cost of deletion that uses this strategy is bounded above by a constant.

4. (15 *points*) Bit-reversed binary counter.

Consider a ***bit-reversal permutation*** on an input array $B[0..n-1]$ whose length is $n = 2^k$ for some nonnegative integer k. This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index a as a k-bit sequence $(a_{k-1}, a_{k-2}, \ldots, a_0)$, where

$$a = \sum_{i=0,\ldots,k-1} a_i 2^i.$$

We define

$$\text{rev}_k(a_{k-1}, a_{k-2}, \ldots, a_0) = (a_0, a_1, \ldots, a_{k-1}).$$

Thus we have

$$\text{rev}_k(a) = \sum_{i=0,\ldots,k-1} a_{k-i-1} 2^i.$$

Note that

$$\text{rev}_k(\text{rev}_k(a)) = a.$$

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\text{rev}_k(3) = 12$, since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12. Therefore, to get the bit-reversal permutation of $B[0..n-1]$, we need to swap $B[3]$ with $B[12]$.

(a) (4 points) Given a function rev_k that runs in $O(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a "bit-reversed counter" and a procedure BITREVERSEDINCREMENT that, when given a bit-reversed-counter value a, produces $\text{rev}_k(\text{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at 0, then successive calls to BITREVERSEDINCREMENT produce the sequence

0000, 1000, 0100, 1100, 0010, 1010, … = 0, 8, 4, 12, 2, 10, ….

This means, to get the bit-reversal permutation of $B[0..n-1]$, we need to swap $B[0]$ with $B[0]$, $B[1]$ with $B[8]$, $B[2]$ with $B[4]$, $B[3]$ with $B[12]$, $B[5]$ with $B[10]$, ….

That is, BITREVERSEDINCREMENT$(a) = \text{rev}_k(\text{rev}_k(a) + 1)$. At the beginning the counter $a = 0$ (so we swap B[0] with B[0]). Then the 1st call gets the counter a to be BITREVERSEDINCREMENT$(0) = \text{rev}_k(\text{rev}_k(0) + 1) = 8$ (so we swap B[1] with B[8]). The 2nd call gets the counter a to be BITREVERSEDINCREMENT$(8) = \text{rev}_k(\text{rev}_k(8) + 1) = 4$ (so we swap B[2] with B[4]). The 3rd call gets the counter a to be BITREVERSEDINCREMENT$(4) = \text{rev}_k(\text{rev}_k(4) + 1) = 12$ (so we swap B[3] with B[12]). ...

(b) (4 points) Suppose we store the bit-reversed counter in an array $A[0..k-1]$ of bits, that is,

$$a = \sum_{i=0,\ldots,k-1} A[i] 2^i.$$

Describe an implementation of the BITREVERSEDINCREMENT procedure on A that allows the bit-reversal permutation on an n-element array to be performed in time $O(n)$. You may assume that, given the bit-array A, the element $B[a] = B[A]$ can be accessed directly in $O(1)$ (no need to explicitly compute a).

(c) (7 points) Describe the $O(n)$-time bit-reversal permutation using the BITREVERSEDINCREMENT procedure in (b). Justify the running time.

5. (*15 points*) Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let

$$k = \lceil \log(n + 1) \rceil,$$

and let the binary representation of n be $(n_{k-1}, n_{k-2}, \ldots, n_0)$. We have k sorted arrays $A_0, A_1, \ldots, A_{k-1}$, where for $i = 0, 1, \ldots, k - 1$, the length of array A_i is 2^i. Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore

$$\sum_{i=0,\ldots,k-1} n_i 2^i = n.$$

Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

(a) (4 points) Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.
(b) (7 points) Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.
(c) (4 points) Describe how to perform the DELETE operation for this data structure. Analyze its worst-case running time.

6. (*10 points*) BFS on graphs

(a) (5 points) Consider an *undirected* graph with 7 vertices: {1, 2, …, 7}, and the following 10 *undirected* edges:

{ (1, 2), (1, 3), (1, 4), (2, 3), (2, 4),
(2, 5), (3, 4), (4, 5), (5, 6), (5, 7) }.

Draw the breath-first tree after running breadth-first search on this graph with source vertex 2.

(b) (5 points) Consider a *directed* graph with 7 vertices: {1, 2, …, 7}, and the following 10 *directed* edges:

(1, 2), (1, 3), (1, 4), (2, 3), (2, 4),
(2, 5), (3, 4), (4, 5), (5, 6), (5, 7).

Draw the breath-first tree after running breadth-first search on this graph with source vertex 2.

7. (*10 points*) Shortest-paths in undirected graphs

We consider undirected graphs (with no weights). Often there are multiple shortest paths between two nodes of a graph. For example, in a graph with vertices {1, 2, 3, 4, 5} and edges

{(1, 2), (1, 3), (2, 4), (3, 4), (4, 5)},

there are two shortest paths from 1 to 5:

(1, 2, 4, 5), and
(1, 3, 4, 5).

Describe a linear-time algorithm such that, given an undirected, unweighted graph and two vertices u and v, the algorithm counts the number of distinct shortest paths from u to v. Justify its correctness and the running time.